Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 2591 – 2594

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Harnessing GPU power from high-level libraries: eigenvalues of integral operators with SLEPc

Jose E. Roman[a,*], Paulo B. Vasconcelos[b]

[a] *D. Sistemes Informàtics i Computació, Universitat Politècnica de València, Camí de Vera s/n, 46022 València, Spain*
[b] *Centro de Matemática da Universidade do Porto and Faculdade de Economia da Universidade do Porto, Rua Dr. Roberto Frias s/n, 4200-464 Porto, Portugal*

Abstract

We consider the approximation of eigenpairs of large-scale matrices arising from the discretization of integral operators with weakly singular function kernels. Efficient and fast solvers to numerically approximate the sought eigenpairs are required. For this, we would like to exploit the computational power of modern graphical processing units (GPUs), and we are interested in doing this from high-level libraries. We show how to use the CUDA add-on in the SLEPc/PETSc libraries to tackle this problem and illustrate our results on a radiative transfer problem in astrophysics. The CUDA-accelerated codes achieve considerable speedups versus the CPU counterparts on the same problem.

Keywords: Numerical libraries, GPU computing, sparse eigensolver, integral operators
2010 MSC: 68W15, 65F15

1. Introduction

Numerical methods to solve large dimensional problems are iterative and their performance depend on the data structure used. It is known that the performance of sparse matrix iterative solvers on GPUs is limited by memory bandwidth, thus the increase in performance when computing eigenvalues and eigenvectors on these systems cannot achieve the remarkable gains usually announced for the dense case, for instance.

In this work we try to understand to which extent this limitation can be stretched. Furthermore, to tackle more complex problems, the researcher/developer should not have to take care of low-level details related to the hardware, whenever possible, and still get reasonable performance. Writing CUDA code directly is a very specialized task. The availability of state-of-the-art libraries capable of performing computations on GPUs is thus of great importance. Even more important is the possibility to have a smooth transition for codes that are already available based on high-level toolkits such as PETSc. Moreover, we would like to assess how difficult it is to plug customized, user-defined GPU kernel code into a computation carried out with SLEPc/PETSc. In our case, we will make use of GPU kernels for the computation of matrix entries, as well as to circumvent the problem of poor performance of sparse matrix-vector products by replacing this operation with a specific band-matrix kernel.

To our best knowledge there are no reports on GPU implementation for the computation of eigenpairs related with integral operators.

*Corresponding author. Tel.: +34-963877356; fax: +34-963877359.
E-mail address: jroman@dsic.upv.es.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.447

2. Problem formulation

We consider the following integral operator $T : X \to X$, issued from a real application [1], defined by

$$(T\varphi)(\tau) = \frac{\varpi}{2} \int_0^{\tau^\star} \int_1^\infty \frac{e^{-|\tau-\tau'|\mu}}{\mu} d\mu\varphi(\tau')d\tau, \quad \tau \in \left[0, \tau^\star\right], \tag{1}$$

which depends on the albedo, $\varpi \in [0, 1]$, and are interested in the eigenvalue problem $T\varphi = \lambda\varphi$ with $\lambda \in \mathbb{C}$, non-null, and $\varphi \neq 0$ defined in X. This problem can be computationally solved by a discretization mechanism, for instance by projecting on a finite dimensional subspace X_n to obtain an algebraic eigenvalue problem $A_n x_n = \theta_n x_n$ of dimension n, where A_n is the restriction of the projected operator to X_n. Further details can be found in [5].

3. Numerical libraries and experiment setup

Among new programming models, CUDA (Compute Unified Device Architecture), and related tools, is playing an important role due to the wide availability of NVIDIA GPU devices. Since writing GPU kernels can be quite difficult, it is recommended to use available utilities for commonly required functionality such as linear algebra operations. Thrust, CUBLAS and CUSP provide high-level interfaces for GPU programming. The first makes common CUDA operations concise and readable while the second is a CUDA implementation of the BLAS (Basic Linear Algebra Subprograms).The latter targets sparse linear algebra and uses functionalities from Thrust.

PETSc (Portable, Extensible Toolkit for Scientific Computation) [2], is a software framework initially designed for the solution of partial differential equations, providing a set of data structures and routines that can be used for the parallel (as well as serial) solution of many numerical problems in scientific and industrial applications. Since version 3.2, PETSc incorporates support for NVIDIA GPUs by means of Thrust and CUSP, performing vector operations and matrix-vector products.

SLEPc (Scalable Library for Eigenvalue Problem Computations) [3], is a software package built on top of PETSc and extends it with all the functionalities necessary for the solution of eigenvalue problems. Basic support for GPUs was included in version 3.2 of the library. Since direct solvers and preconditioners are not yet prepared to run on the GPU, some SLEPc functionality such as the shift-and-invert transformation is restricted to be run on the CPU only. Also, a great performance boost should not be expected by running the eigensolvers on the GPU instead of the CPU, due to the limited performance of the sparse matrix-vector product. Higher Gflop/s rates can be achieved by using customized kernels, as will be illustrated in section 4.

The numerical approach to the algebraic eigenproblem depends, naturally, on the matrix A_n properties. One can observe a decay in magnitude in the matrix entries from the diagonal, relating grid-points farthest from each other. Since this decay is very steep, either a sparse structure approach or a band one can be considered. Hence, both data structures are to be explored on the numerical experiments. The problem is difficult to solve since for larger values of n the eigenvalues tend to be tightly clustered. This difficulty can be overcome by using the shift-and-invert strategy, but we do not consider this here since it requires computing matrix factorizations or, in the sparse case, using efficient implementations of preconditioners (currently lacking on the GPU). A less effective alternative to cope with this difficulty is to increase the dimension of the basis of the search space. For the problems tested in the following section a value of 48 was considered in all runs, except the largest case ($n = 128000$) which used a basis of 96 vectors.

The tests have been executed on a Linux workstation equipped with an Intel Core i7 950 processor at 3,06 GHz with 8 MB of L3 cache memory and 8 GB of main memory (4 cores with hyper-threading technology, a total of 8 virtual processors). The computing platform consists on two NVIDIA Tesla C2050, based on the Fermi GPU architecture, with 448 cores and 3 GB GDDR per GPU. Only one GPU has been used for the experiments. The software configuration is based on Ubuntu Linux 10.04 LTS, with GCC 4.4.3 (the GNU C compiler), PETSc 3.3, SLEPc 3.3, CUDA 4.0 and CUSP 0.2.0.

4. Numerical experiments

For the numerical tests, a combination of data structures for matrix storage and different approaches for the generation and solution phases are considered. It is worth mentioning that, apart from matrix-vector products

carried out on the GPU as discussed below, our implementation performs many other computations (mainly vector operations such as orthogonalization) on the GPU as well. In the experiments discussed below, the cost of these operations amount up to 15% of the overall cost at most, so our analysis focuses on the matrix-vector product.

We consider five alternative storage schemes. Scheme (*A*), the baseline, performs both phases (generation and resolution) on the CPU using the standard format within PETSc, the general sparse AIJ format (compressed sparse row, CSR). Schemes (*B*) and (*C*) perform the matrix generation phase on the CPU, maintaining a copy of the matrix on the GPU, and solve the problem on the GPU. The two schemes differ in the format used to store the matrix on the GPU and hence the method for performing matrix-vector products. The former, scheme (*B*), uses CUSP for sparse matrix computations on the GPU via PETSc's specialized matrix class AIJCUSP, being the internal data structure as in (*A*). The matrix elements are assembled in the standard PETSc way on the CPU but only ready for use after being copied to the device. In the latter, (*C*), the generation is done as in (*B*) but the copy to the device is performed as a (dense) banded matrix and the matrix-vector operations are carried-out by a CUBLAS subroutine, cublasDgbmv. The fourth scheme, (*D*), performs all computations on the GPU. All data is generated directly on the GPU through a call to a CUDA kernel that employs a CUDA thread for each of the matrix entries. The matrix-vector operations required for the Krylov-Schur solution method are performed similarly to scheme (*C*), that is, with CUBLAS. The last scheme, (*E*), is a variation of (*D*) where the storage on the GPU follows a block-banded format rather than the plain band storage required by CUBLAS. In this case, matrix-vector products are computed via a specialized GPU kernel.

For the generation phase, the difference between the two kernels on schemes (*D*) and (*E*) is just the arrangement of matrix elements in memory (band vs block-band). These kernels are organized in such a way that CUDA thread blocks are arranged in tiles, where the tile width and height are tunable parameters (the only limit is the number of physical cores in the GPU - 448 in our case).

For the matrix-vector product on the GPU, three options were considered: (i) CUSP with a CSR sparse format (this alternative is very simple, since it can be accomplished by using PETSc options), (ii) CUBLAS with a band storage (this solution is very effective, because it is much cheaper in terms of memory requirements than the previous one, and also it can achieve a high Gflop/s rate), (iii) Customized kernel with a block-band storage (implementation of this strategy is even more trickier than the CUBLAS one, since in addition the programmer has to provide a customized kernel for matrix-vector multiplication).

Table 1. Elapsed time (seconds) for the generation and solver phases, with $\tau^\star = 1000$, n ranging from 4000 to 64000, for schemes $(A): CPU^{CSR}_{gen.}+CPU^{CSR}_{sol.}$, $(B): CPU^{CSR}_{gen.}+GPU^{CSR}_{sol.}$, $(C): CPU^{CSR}_{gen.}+GPU^{band}_{sol.}$, $(D): GPU^{band}_{gen.}+GPU^{band}_{sol.}$ and $(E): GPU^{block-band}_{gen.}+GPU^{block-band}_{sol.}$ (`tile_width` = 16×16 and `num_threads` = 64).

	n	4000	8000	16000	32000	64000
	generation	15.2	61.4	245.2	981.1	3921.3
(A) :	solver	2.6	9.1	32.8	121.9	489.0
	total	17.8	70.5	278.0	1102.0	4410.3
	generation	15.2	61.4	245.2	982.0	3931.8
(B) :	solver	1.8	2.8	5.6	14.4	n/a
	total	17.0	64.2	250.7	996.4	n/a
	generation	15.2	61.4	245.2	981.1	3921.3
(C) :	solver	2.1	3.9	6.2	14.5	47.1
	total	17.3	65.3	251.4	995.6	3968.4
	generation	0.1	0.2	0.5	1.7	6.6
(D) :	solver	2.1	5.2	5.9	13.5	42.1
	total	2.2	5.4	6.4	15.3	48.7
	generation	0.1	0.1	0.5	1.7	6.7
(E) :	solver	1.8	2.7	4.8	10.8	32.7
	total	1.9	2.8	5.3	12.5	39.4

We now present timing data for the proposed methods for $\varpi = 0.75$ and $\tau^\star = 1000$. Table 1 presents computing times for the five proposed methods in problem sizes ranging from 4000 to 64000. Times are split into matrix generation and solution of the problem (eigenvalue computation with Krylov-Schur). For the generation phase it is evident the enormous gain obtained from performing the evaluation of the matrix entries directly on the device, instead of computing them on the CPU. There is a negligible overhead in schemes (*B*) and (*C*), with respect to (*A*), related to the copy of the matrix to the GPU. Also, there is no significant difference between the sparse and

band approaches in term of matrix generation in the GPU. For the solver phase, the gains achieved by performing device computations compared to CPU are clear. Using CUBLAS for the matrix-vector product on the device induces a slight increase in elapsed time with respect to its sparse counterpart, scheme (C) versus scheme (B), but it needs less memory (the $n = 64000$ case exceeds available device memory with CUSP). Although scheme (D) also deals with a band data structure, there are small gains by not having a copy on the host. Finally, the optimized custom kernel of (E) demonstrates its competitiveness with respect to (C) and (D).

The generation dominates the overall costs whenever the CPU is used, whereas for the GPU it is the solver phase that requires more time. Around a 100−fold improvement is the overall gain for the larger problem. Yet, and although using high-level libraries with CUDA support, these gains are still at a great developing cost, since the parts of the code that had to be rewritten are not negligible.

The custom kernel of scheme (E) is 25% faster than the CUBLAS option. In addition, this kernel can be further optimized by tuning the parameters that can have impact on performance. In Tables 2 and 3 we report on varying values of the tile width and `num_threads`, respectively, for a problem size $n = 128000$ (with $\tau^\star = 4000$). In the case of the tile width, all tested combinations show good results, with minor differences among them. In contrast, the number of threads per block is more critical for achieving good performance in the solver phase, being 64 the best value for these experiments.

Table 2. Elapsed time (seconds) for the generation phase, with $\tau^\star = 4000$, $n = 128000$, for scheme (E) : $GPU^{blockband}_{gen.} + GPU^{band}_{sol.}$ (several `tile_width` and `num_threads` = 64).

	10×10	16×16	20×20	14×16	14×32	32×14
(E) :	9.4	7.7	8.2	7.5	7.7	8.05

Table 3. Elapsed time (seconds) for the solver phase, with $\tau^\star = 4000$, $n = 128000$, for scheme (E) : $GPU^{blockband}_{gen.} + GPU^{band}_{sol.}$ (`tile_width` = 16×16 and several `num_threads`).

	16	32	64	128
(E) :	247.2	189.1	175.9	183.4

5. Conclusions

We presented some GPU approaches to solve an eigenvalue problem coming from an astrophysics application, taking advantage from high-level libraries. With this work, we also show how SLEPc provides GPU support, profiting from the CUDA support developed in PETSc. More interestingly, customized kernels can be implemented and integrated in the solution scheme to obtain better performance.

Results show slightly more than 100× speedup when the problem is completely solved (generation phase and eigenvalue computations) by the GPU with respect to the CPU approach. Maintaining a copy at the GPU of the data generated on the CPU, greatly facilitates the coding of a numerical implementation; but in order to explore as much as possible the power inside the device to achieve high performances, programing directly on the GPU is mandatory. The most time consuming part of the developed solution depends on the approach followed: while the generation phase dominates the required elapsed time on the CPU, it is the solver phase that matters on the GPU.

References

[1] Ahues, M., d'Almeida, F.D., Largillier, A., Titaud, O., Vasconcelos, P.B.: An L^1 refined projection approximate solution of the radiation transfer equation in stellar atmospheres. J Comput Appl Math 140 (2002) 13–26.

[2] Balay, S., Brown J., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.3, Argonne National Laboratory (2012)

[3] Hernandez, V., Roman, J.E., Vidal, V.: SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. ACM Trans. Math. Soft. 31(3) (2005) 351–362

[4] Minden, V., Smith, B.F., Knepley, M.G.: Preliminary implementation of PETSc using GPUs. Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering (2010).

[5] Vasconcelos, P.B., Marques, O., Roman, J.E.: Parallel Eigensolvers for a Discretized Radiative Transfer Problem. Lect Notes Comput Sc 5336 (2008) 336–348.